APPENDIX A

PACKET AND MESSAGE FORMATS:

In one embodiment, tokenbeat packets are limited to a maximum of 448 bytes per packet. This allows for Tokenbeat to pass over physical layers with relatively small Maximum Transmit Unit unfragmented.

Each packet may contain multiple messages. In this way we can economize on the overhead of the underlying transport headers, and Tokenbeat packet headers. This is expected to be the normal mode of operation. However, should it be necessary, a node may create and send a new packet. This provides for an out-of-band communication facility.

Tokenbeat rings are limited to 254 or fewer machines. This is due to the fact that the each machine receives a 1 byte ring ID. The address 0x00 is reserved for stations which have not yet received an address. The address 0xFF is the broadcast address. The following diagram illustrates the format of a Tokenbeat packet. It is in network (big-endian) byte order.

```
               1 1
0       7 8    5 6
 +--------+--------+--------//
 |  ver   | mcount | msgs...
 +--------+--------+--------//
```

ver:

This is the protocol version. The current version is 1.

mcount:

This identifies the number of messages contained in the current packet. We are limited to 255 or fewer messages per packet.

The following diagram illustrates the format of a Tokenbeat message:

```
                 1 1      2 2      3 3
0        7 8     5 6      3 4      1 2
+--------+--------+--------+--------+--------//
| ttl    |  dst   |  src   |  type  | data...
+--------+--------+--------+--------+--------//
```

ttl:

This is the Time To Live field. Messages with a zero TTL must not be passed on, but they must be processed as valid messages. The use of the TTL field allows messages to die even if the target station cannot be found and the source station does not remove the message from the circulating packet.

Each node in the ring decrements the TTL field as a message passes through it, even if the message is not addressed to that particular station.

dst:

The Tokenbeat address of the destination. This may be an individual station ID or 0xFF indicating a broadcast message.

src:

This is the Tokenbeat address of the originating station. In the event that the originating station does not yet have a Tokenbeat address, it should use the address 0x00.

type:

This field identifies the message type. It may be one of the following:

TB_INVALID (0)	This special value allows us to watch for invalid Tokenbeat messages. It will identify misbehaving stacks which may be sending out zeroed-out messages.

TB_OKAY	(1) The source station is operating normally. Additionally, it may be inferred that the ring up to the current receiving station is healthy.
TB_HOSTUPENQ	(2) This message indicates that a new host has entered the ring.
TB_HOSTUPACK	(3) This message is the appropriate response to a TB_HOSTUPENQ message.
TB_RINGPURGE	(4) In the event that a Tokenbeat station detects a break in the ring, it will broadcast a TB_RINGPURGE whereupon all active stations will reinitialize their Tokenbeat stack and the ring will rebuild itself. A TB_RINGPURGE message may be used any other time it might be useful to force the ring to rebuild.
TB_CONNENQ	(6) This message allows the source address to query the destination about the status of network connections.
TB_CONNSTAT	(7) This message type allows the source station to identify the status of a network connection.
TB_PING	(8) This allows for a station to inquire about stations currently active on the ring. The appropriate response to a TB_PING messages is a TB_ECHO message.
TB_ECHO	(9) This message is used to indicate that a station is alive and active on the Tokenbeat ring.
TB_HOSTID	(10) This message can be used to assign a Tokenbeat station ID to station.

TB_HOSTDOWN (11) This message is used to indicate that a station is going offline.

data:

This field is optional. It contains any data that may be pertinent to the current message in a message specific format.

MESSAGE FORMATS:

This section will describe the formats of each of the Tokenbeat message types.

TB_INVALID:

This special message type should never be used. In the event that a Tokenbeat message is found to contain the TB_INVALID type, the rest of the packet should be discarded.

TB_OKAY:

This message indicates two things:

1. The originating Tokenbeat station is operating normally.

2. The ring between the current station and the originating station is functional.

By extension, if the originating station gets its message back, the ring is whole and healthy.

There is no need to respond to a TB_OKAY message. It is merely passed on the same as any message.

There is no data field.

TB_HOSTUPENQ:

This message indicates that the originating station is alive.

The data field of this message is the lower layer address of the source station, in network byte order.

As the newly active station is not a part of the ring, this message will be broadcast. Every station receiving this message should return a TB_HOSTUPACK message.

TB_HOSTUPACK:

This message is used to inform a newly active station about other stations on the ring. Upon receiving a TB_HOSTUPENQ message, all active stations must send a TB_HOSTUPACK.

The data field of this message contains the lower layer address of the responding station in network byte order.

TB_RINGPURGE:

This message indicates that there has been some sort of catastrophic event. The ring must be rebuilt.

This message is broadcast. All stations receiving it must reinitialize their Tokenbeat stack. That is to say, they must ``forget" their upstream and downstream neighbors, etc., and send out a TB_HOSTUPENQ message as they did when they first inserted into the ring.

This is primarily for use in the event that a host goes down (thus breaking the ring).

There is no data field.

TB_OKAY:

This message indicates that the originating station is operating normally.

Typically, TB_OKAY messages will be circulated by the ring master in the case that there is no other event to report. This keeps the token circulating.

There is no data field.

TB_CONNENQ:

This message allows a station to query the status of network connections on the destination station.

The data field consists of a set of bit masks. The first is the lower layer address to be applied to the address of the station receiving the message. Additional fields may be added to further clarify connections. I.e., in a TCP/IP environment, the first part may be 4 bytes to mask the IP address and the second part may be 2 bytes to mask the TCP port number.

As an example, let assume our underlying network were TCP/IP. If the IP mask were 0xC0A801FF, it would match every host in the class C subnet 192.168.1.0. If the port mask were 0x03FF, it would match every port under 1024. Thus the pair would match every connection to a host in 192.168.1.0 on a port less than 1024.

If the receiving station has any connections to addresses matching these masks, it is expected to issue a TB_CONNSTAT message. This provides a mechanism for rebuilding a map of network connections to nodes in the ring. This is useful in the event that we have a central authority which must have this information.

This allows us to use centralized algorithms without catastrophic failure should the central authority fail.

TB_CONNSTAT:

This message is used to inform another station of the state of connections.

Each message contains information about only one connection. Multiple connection states require multiple messages.

The data field consists of the local address (network or physical) followed by any additionally necessary information in a fashion similar to TB_CONNECNQ, followed by the remote address followed by any further information necessary to identify the remote endpoint.

TB_PING:

This message allows a station to inquire about other stations in the ring. The appropriate response is a TB_ECHO message.

There is no data field.

TB_ECHO:

This is the response to a TB_PING message. The data field consists of the lower layer address of the responding station in network byte order.

Note that TB_PING/TB_ECHO pairs may be used to map lower layer addresses to Tokenbeat station numbers. In this way, we can build a map of the entire Tokenbeat ring, not just in terms of Tokenbeat station IDs, but in terms of lower layer networking as well.

TB_HOSTID:

This message assigns a Tokenbeat station ID to a particular station. It is typically only sent by the Tokenbeat ring master.

The data field consists of the lower layer address of the station being assigned a new ID followed by the new ID, in network byte order.

Typically this will be addressed to Tokenbeat station 0xFF (the broadcast) so that the station assigning the address does not need to determine if the station being assigned a new address already has one.

TB_HOSTDOWN:

This message indicates that a station is going offline and that any other stations in the ring should take appropriate action.

The data field consists of the lower layer address of the station going offline.

Appendix B

This section evaluates experimental results obtained from a prototype of the SASHA architecture. We consider the results of tests in various fault scenarios under various loads.

Our results demonstrate that in tests of real-world (and some not-so-real-world) scenarios, our SASHA architecture provides a high level of fault tolerance. In some cases, faults might go unnoticed by users since they are detected and masked before they make a significant impact on the level of service. Our fault-tolerance experiments are structured around three levels of service requested by client browsers: 2500 connections per second (cps), 1500 cps, and 500 cps. At each requested level of service, we measured performance for the following fault scenarios: no-faults, a dispatcher server faults, three server faults, and four server faults. Figure 3 summarizes the actual level of service provided *during the fault detection and recovery interval* for each of the failure modes. In each fault scenario, the final level of service was higher than the level of service provided during the detection and recovery process. The rest of this section details these experiments as well as the final level of service provided after fault recovery.

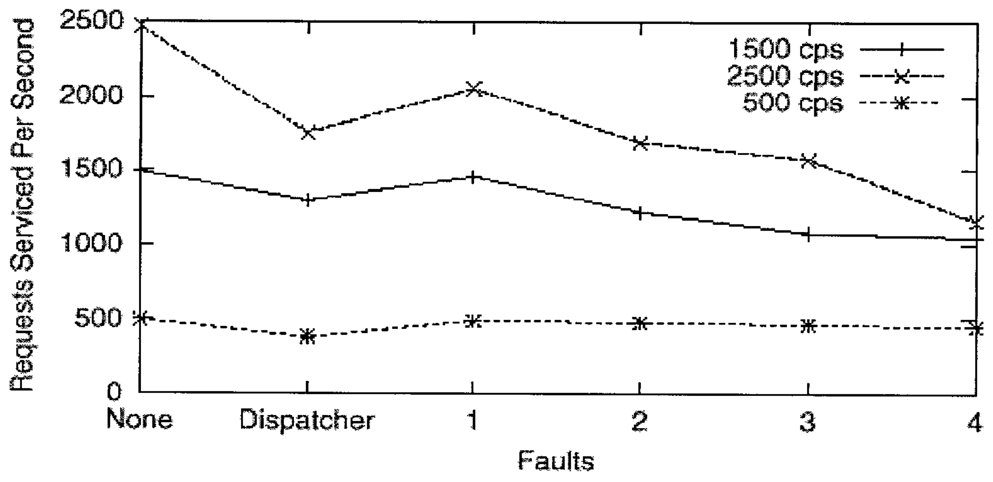

Figure 3: System performance, in requests serviced per second, during fault detection and recovery for three levels of requested service: 2500 connections per second (cps), 1500 cps, and 500 cps.

2,500 Connections Per Second

In the first case, we examined the behavior of a cluster consisting of five server nodes and the K6-2 400 dispatcher. Each of our five clients generated 500 requests per second. This was the maximum sustainable load for our clients and servers, though dispatcher utilization suggests that it may be capable of supporting up to 3,300 connections per second. Each test ran for a total of 30 seconds. This short duration allows us to more easily discern the effects of node failure. Figure 3 shows that in the base, non-faulty, case we are capable of servicing 2,465 connections per second.

In the first fault scenario, the dispatcher node was unplugged from the network shortly after beginning the test. We see that the average connection rate drops to 1,755 connections per second (cps). This is to be expected, given the time taken to purge the ring and detect the dispatcher's absence. Following the startup of a new dispatcher, throughput returned to 2,000 cps, or of the original rate. Again, this is not surprising as the servers were operating at capacity previously and thus losing one of five nodes drops the performance to 80% of its previous level.

Next we tested a single-fault scenario. In this case, shortly after starting the test, we removed a server from the network. Results were slightly better than expected. Factoring in the connections allocated to the server before its loss was detected and given the degraded state of the system following diagnosis, we still managed to average 2,053 connections per second.

In the next scenario, we examined the impact of coincident faults. The test was allowed to get underway and then one server was taken offline. After the system had detected and diagnosed, the next server was taken offline. Again, we see a nearly linear performance decrease in performance as the connection rate drops to 1,691 cps. The three fault scenario was similar to the two fault scenario, save that performance ends up being 1,574 cps. This relatively high performance–given that there are, at the end of the test, only two active servers–is most likely due to the fact that the state of the server gradually degrades over the course of the test. We see similar behavior with a four fault scenario. By the end of the four fault test, performance had stabilized at just over 500 cps, the maximum sustainable load for a single server.

1,500 Connections Per Second

This test was similar to the 2,500 cps test, but with the servers less utilized. This allows us to observe the behavior of the system in fault-scenarios where we have excess server capacity. In this configuration, the base, no-fault, case shows 1,488 cps. As we have seen above, the servers are capable of servicing a total of 2,500 cps, therefore the cluster is only 60% utilized. Similar to the 2,500 cps test, we first removed the dispatcher midway through the test. Again performance drops, as expected–to 1,297 cps in this case. However, owing to the excess capacity in the clustered server, by the end of the test, performance had returned to 1,500 cps. For this reason, the loss and election of the dispatcher seems less severe, relatively speaking, in the 1,500 cps test than in the 2,500 cps test.

In the next test, a server node was taken offline shortly after starting the test. We see that the dispatcher rapidly detects and masks this. Total throughput ended up at 1,451 cps. The loss of the server was nearly undetectable.

Next, we removed two servers from the network, similar to the two-fault scenario in the 2,500 cps environment. This makes the system into a three-node server operating at full capacity. Consequently, it has more difficulty restoring full performance after diagnosis. The average connection rate comes out at 1,221 cps.

In the three fault scenario, similar to our previous three fault scenario, we now examine the case where the servers are overloaded after diagnosis and recovery. This is reflected in the final rate of 1,081 cps. Again, while the four fault case has relatively high average performance, by the end of the test, it was stable at a little over 500 cps, our maximum throughput for one server.

500 Connections Per Second

Following the 2,500 and 1,500 cps tests, we examined a 500 cps environment. This gave us the opportunity to examine a highly under utilized system. In fact, we had an "extra" four servers in this configuration since one server alone is capable of servicing a 500 cps load. This fact is reflected in all the fault scenarios. The most severe fault occurred with the dispatcher. In that case, we lost 2,941 connections to timeouts. However, after diagnosing the failure and electing a new dispatcher, throughput returned to a full 500 cps.

In the one, two, three, and four server-fault scenarios, the failure of the server nodes is nearly impossible to see on the graph. The final average throughput was 492.1, 482.2, 468.2, and 448.9 cps as compared with a base case of 499.4. That is, the loss of four out of five nodes over the course of thirty seconds caused a mere 10% reduction in performance.

Extrapolation

We have demonstrated that given the hardware available at the time of the 1998 Olympic Games (400 MHZ *x86*), an application-space solution would have been adequate to service the load. To further test the hypothesis that application-space dispatchers operating on commodity systems provide more than adequate performance, we looked at a dispatcher that could have been deployed at the time of the 1996 Olympic Games versus the 1996 Olympic web traffic. Operating under the assumption that the number and type of web servers is not particularly important (owing to the high degree of parallelism, performance grows linearly in this architecture until the dispatcher or network are saturated), the configuration remained the same as previous tests with the exception that the dispatcher node was replaced with a Pentium 133.

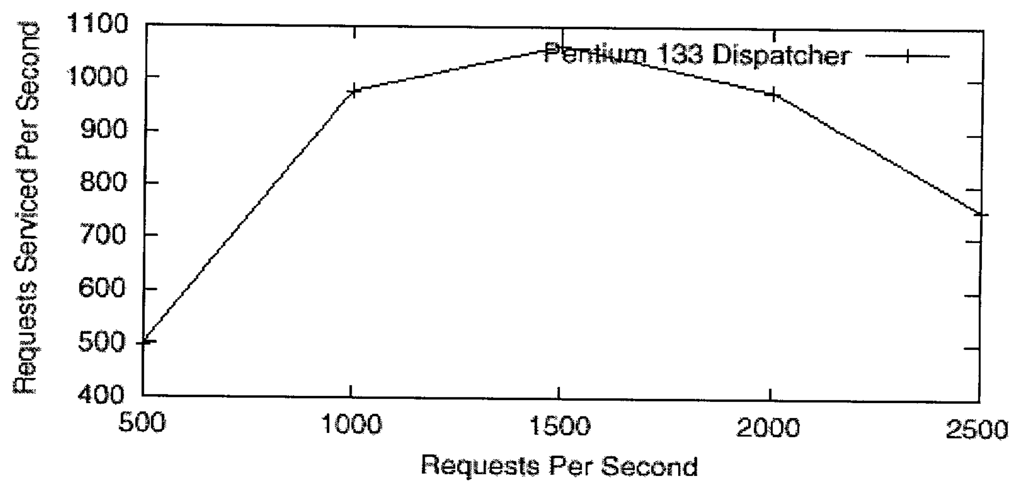

Figure 4: Performance with a Pentium 133 dispatcher.

As we see in Figure 4, at 500 and 1,000 cps, we are capable of servicing all the requests. By the time we reach 1,500 cps, we can service just over 1,000. 2,000 and 2,500 cps actually see worse service as the dispatcher becomes congested and packets are dropped, nodes must retransmit, and traffic flows less smoothly. The 1996 games saw, at peak load, 600 cps. That is, our capacity to serve is 1.8 times the actual peak load. In similar fashion, we believe our 1998 vintage hardware is capable of dispatching approximately 3,300 connections per second, again about 1.8 times the actual peak load. While we only have two data points from which to extrapolate, we conjecture that COTS systems will continue to provide performance sufficient to service even the most extreme loads easily.